Data Preparation

Rita P. Ribeiro

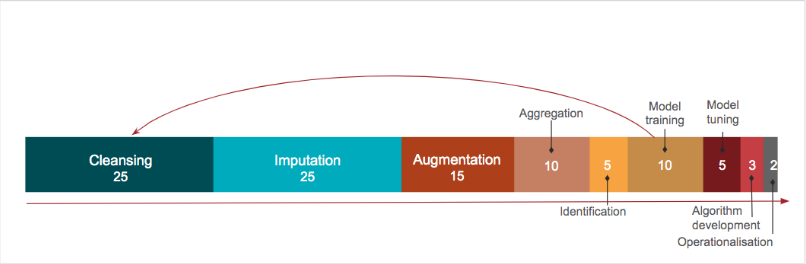
Machine Learning - 2021/2022

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

[dcc] DEPARTAMENTO DE CIÊNCIA DE COMPUTADORES
FACULDADE DE CIÊNCIAS DA UNIVERSIDADE DO PORTO

Data Preparation

- The majority of time taken by any data mining project is spent in data preparation
  - e.g. importing, manipulating, cleaning, transforming, augmenting



"Average time allocated to machine-learning project tasks" %of total
SOURCE: Cognilytica, 2019

Data

What is Data?

Collection of data objects (cases) and their attributes (features)

- Attribute: a property or characteristic of an object
  - date, country, temperature, precipitation
- Object: described by a collection of attributes
- It can be structured (e.g. data table) or non-structured (e.g. text)
- It can have non-dependency or dependency between objects (e.g. time, space)

What is Data? (cont.)

Types of data sets

- Nondependency-oriented data
  - the cases do not have any dependencies between them
  - examples: simple data tables, text
- Dependency-oriented data
  - implicit or explicit relationships between cases
  - examples: time series, discrete sequences, spatialtemporal data, network and graph data.

Data: Types of Attributes

- Categorical / Qualitative Attributes
  - Nominal: there is no relationship between the values
    - name, gender, patient id
  - Ordinal: there is an order between the values, but no mathematical operation can be performed on them
    - size $\in \{small, medium, large\}$
- Numeric / Quantitative Attributes
  - Discrete: finite or countably infinite set of values for which differences are meaningful
    - temperatures in Celsius, calendar dates, event duration in minutes
  - Continuous: inifinite set of values that represent the absolute numbers
    - number of visits to the hospital, distance, income

Data: Important Characteristics

- Dimensionality (i.e. number of attributes)
  - high dimensional data brings several challenges
- Sparsity
  - presence attributes
- Resolution
  - patterns depend on the scale
- Size
  - type of analysis may depend on size of data

Data Preparation

- Typically, data analysis tasks use source data sets stored in tabular format.
    - datasets are bi-dimensional structures (e.g. table)

- How can we **import data** from different sources and / or formats?
- How can we easily **manipulate the data**?
- How can we **transform the data**?

Data Wrangling

- Process of transforming and mapping data from one “raw” data form into another format appropriate for analytics.
- Main steps
  - discovering
  - structuring
  - cleaning
  - enriching
  - validating
  - publishing
- Goal: attain quality and useful data.

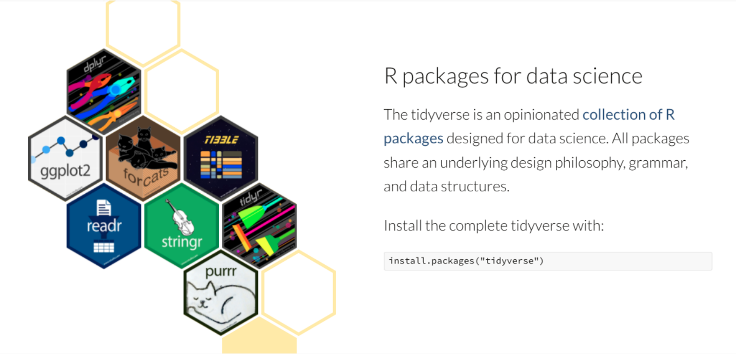
Data Wrangling in R

Data Objects

- Tidy data:
  - every column is variable
  - every row is an observation
  - every cell is a single value
- Data objects: `tibbles`
  - `int`: integers.
  - `dbl`: doubles, or real numbers.
  - `chr`: character vectors, or strings.
  - `dttm`: date-times (a date + a time).
  - `lgl`: logical, vectors that contain only TRUE or FALSE.
  - `fctr`: factors, i.e. categorical variables with fixed possible values.
  - `date`: dates.

tidyverse - R packages for Data Science

- **tidyverse** - R packages for Data Science



`tidyverse` - R packages for Data Science (cont.)

- **`readr`**: provides a fast and friendly way to read rectangular data (e.g. csv)
- **`tidyr`**: helps you create tidy data
- **`stringr`**: cohesive set of functions designed to make working with strings as easy as possible.
- **`forcats`**: provides a suite of tools that solve common problems with factors (categorical variables handled in R)
- **`dplyr`**: grammar of data manipulation
- **`ggplot2`**: grammar of graphics

Package `readr`: importing data

"dummy.csv"

```
ID, Name, Height
23424, Ana, 1.60
11234, Charles, 1.73
77654, Susanne, 1.65
```

```r
ds <- read_csv("dummy.csv")
ds
```

```
## # A tibble: 3 x 3
##      ID Name    Height
##   <dbl> <chr>    <dbl>
## 1 23424 Ana       1.6
## 2 11234 Charles   1.73
## 3 77654 Susanne   1.65
```

Package `readr`: importing data (cont.)

"dummy2.csv"

```
ID; Name; Height
23424; Ana; 1,60
11234; Charles; 1,73
77654; Susanne; 1,65
```

```r
ds <- read_delim("dummy2.csv", delim = ";")
ds
```

```
## # A tibble: 3 x 3
##      ID Name    Height
##   <dbl> <chr>    <dbl>
## 1 23424 Ana        160
## 2 11234 Charles    173
## 3 77654 Susanne    165
```

Package `readr`: importing data (cont.)

"dummy2.csv"

```
ID; Name; Height
23424; Ana; 1,60
11234; Charles; 1,73
77654; Susanne; 1,65
```

```r
ds <- read_delim("dummy2.csv", delim = ";", locale = locale(decimal_mark = ","))
ds
```

```
## # A tibble: 3 x 3
##       ID Name    Height
##    <dbl> <chr>    <dbl>
## 1 23424 Ana       1.6
## 2 11234 Charles   1.73
## 3 77654 Susanne   1.65
```

Package `readr`: importing data (cont.)

"dummy.txt"

```
ID, Name, Height
23424, Ana, ?
11234, Charles, 1.73
77654, Susanne, 1.65
```

```r
ds <- read_csv("dummy.txt", na = "?")
ds
```

```
## # A tibble: 3 x 3
##      ID Name    Height
##   <dbl> <chr>    <dbl>
## 1 23424 Ana      NA
## 2 11234 Charles   1.73
## 3 77654 Susanne   1.65
```

Data import with the tidyverse : : CHEAT SHEET

readr

Read Tabular Data with readr

read_*(file, col_names = TRUE, col_types = NULL, col_select = NULL, id = NULL, locale, n_max = Inf, skip = 0, na = c("", "NA"), guess_max = min(1000, n_max), show_col_types = TRUE) See **?read_delim**

read_delim("file.txt", delim = "|"**)** Read files with any delimiter. If no delimiter is specified, it will automatically guess.
To make file.txt, run: write_file("A|B|C\n1|2|3\n4|5|NA", file = "file.txt")

read_csv("file.csv"**)** Read a comma delimited file with period decimal marks.
write_file("A,B,C\n1,2,3\n4,5,NA", file = "file.csv")

read_csv2("file2.csv"**)** Read semicolon delimited files with comma decimal marks.
write_file("A;B;C\n1,5;2;3\n4,5;5;NA", file = "file2.csv")

read_tsv("file.tsv"**)** Read a tab delimited file. Also **read_table()**.
read_fwf("file.tsv", fwf_widths(c(2, 2, NA))**)** Read a fixed width file.
write_file("A\tB\tC\n1\t2\t3\n4\t5\tNA\n", file = "file.tsv")

USEFUL READ ARGUMENTS

No header
read_csv("file.csv", col_names = FALSE)

Provide header
read_csv("file.csv",
col_names = c("x", "y", "z"))

Read multiple files into a single table
read_csv(c("f1.csv", "f2.csv", "f3.csv"),
id = "origin_file")

Skip lines
read_csv("file.csv", skip = 1)

Read a subset of lines
read_csv("file.csv", n_max = 1)

Read values as missing
read_csv("file.csv", na = c("1"))

Specify decimal marks
read_delim("file2.csv", locale =
locale(decimal_mark = ","))

Save Data with readr

write_*(x, file, na = "NA", append, col_names, quote, escape, eol, num_threads, progress)

write_delim(x, file, delim = " "**)** Write files with any delimiter.

write_csv(x, file**)** Write a comma delimited file.

write_csv2(x, file**)** Write a semicolon delimited file.

write_tsv(x, file**)** Write a tab delimited file.

One of the first steps of a project is to import outside data into R. Data is often stored in tabular formats, like csv files or spreadsheets.

The front page of this sheet shows how to import and save text files into R using **readr**.

The back page shows how to import spreadsheet data from Excel files using **readxl** or Google Sheets using **googlesheets4**.

OTHER TYPES OF DATA

Try one of the following packages to import other types of files:

- **haven** - SPSS, Stata, and SAS files
- **DBI** - databases
- **jsonlite** - json
- **xml2** - XML
- **httr** - Web APIs
- **rvest** - HTML (Web Scraping)
- **readr::read_lines()** - text data

Column Specification with readr

Column specifications define what data type each column of a file will be imported as. By default readr will generate a column spec when a file is read and output a summary.

spec(x**)** Extract the full column specification for the given imported data frame.

```
spec(x)
# cols(
#   age = col_integer(),
#   sex = col_character(),
#   earn = col_double()
# )
```

age is an integer

sex is a character

earn is a double (numeric)

COLUMN TYPES

Each column type has a function and corresponding string abbreviation.

- **col_logical()** - "l"
- **col_integer()** - "i"
- **col_double()** - "d"
- **col_number()** - "n"
- **col_character()** - "c"
- **col_factor(**levels, ordered = FALSE**)** - "f"
- **col_datetime(**format = ""**)** - "T"
- **col_date(**format = ""**)** - "D"
- **col_time(**format = ""**)** - "t"
- **col_skip()** - "-", "_"
- **col_guess()** - "?"

USEFUL COLUMN ARGUMENTS

Hide col spec message
read_*(file, show_col_types = FALSE)

Select columns to import
Use names, position, or selection helpers.
read_*(file, col_select = c(age, earn))

Guess column types
To guess a column type, read_ *() looks at the first 1000 rows of data. Increase with **guess_max**.
read_*(file, guess_max = Inf)

DEFINE COLUMN SPECIFICATION

Set a default type
```
read_csv(
  file,
  col_type = list(.default = col_double())
)
```

Use column type or string abbreviation
```
read_csv(
  file,
  col_type = list(x = col_double(), y = "l", z = "_")
)
```

Use a single string of abbreviations
```
# col types: skip, guess, integer, logical, character
read_csv(
  file,
  col_type = "_?ilc"
)
```

RStudio

Package `readr`: importing data (cont.)

Import Spreadsheets with readxl

READ EXCEL FILES

read_excel(path, sheet = NULL, range = NULL**)**
Read a .xls or .xlsx file based on the file extension. See front page for more read arguments. Also **read_xls()** and **read_xlsx()**.
read_excel("excel_file.xlsx")

READ SHEETS

read_excel(path, **sheet = NULL)** Specify which sheet to read by position or name.
read_excel(path, sheet = 1)
read_excel(path, sheet = "s1")

excel_sheets(path) Get a vector of sheet names.
excel_sheets("excel_file.xlsx")

To **read multiple sheets:**
1. Get a vector of sheet names from the file path.
2. Set the vector names to be the sheet names.
3. Use purrr::map_dfr() to read multiple files into one data frame.

```
path <- "your_file_path.xlsx"
path %>% excel_sheets() %>%
  set_names() %>%
  map_dfr(read_excel, path = path)
```

READXL COLUMN SPECIFICATION

Column specifications define what data type each column of a file will be imported as.

Use the **col_types** argument of **read_excel()** to set the column specification.

Guess column types
To guess a column type, read_ excel() looks at the first 1000 rows of data. Increase with the **guess_max** argument.
read_excel(path, **guess_max** = Inf)

Set all columns to same type, e.g. character
read_excel(path, col_types = "text")

Set each column individually

```
read_excel(
  path,
  col_types = c("text", "guess", "guess","numeric")
)
```

COLUMN TYPES

logical	numeric	text	date	list
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

- skip
- guess
- logical
- numeric
- text
- date
- list

Use **list** for columns that include multiple data types. See **tidyr** and **purrr** for list-column data.

with googlesheets4

READ SHEETS

read_sheet(ss, sheet = NULL, range = NULL**)**
Read a sheet from a URL, a Sheet ID, or a dribble from the googledrive package. See front page for more read arguments. Same as **range_read()**.

SHEETS METADATA

URLs are in the form:
https://docs.google.com/spreadsheets/d/**SPREADSHEET_ID**/edit#gid=**SHEET_ID**

gs4_get(ss) Get spreadsheet meta data.

gs4_find(...) Get data on all spreadsheet files.

sheet_properties(ss) Get a tibble of properties for each worksheet. Also **sheet_names()**.

WRITE SHEETS

write_sheet(data, ss = NULL, sheet = NULL**)** Write a data frame into a new or existing Sheet.

gs4_create(name, ..., sheets = NULL**)** Create a new Sheet with a vector of names, a data frame, or a (named) list of data frames.

sheet_append(ss, data, sheet = 1**)** Add rows to the end of a worksheet.

GOOGLESHEETS4 COLUMN SPECIFICATION

Column specifications define what data type each column of a file will be imported as.

Use the **col_types** argument of **read_sheet()/ range_read()** to set the column specification.

Guess column types
To guess a column type read_sheet()/ range_read() looks at the first 1000 rows of data. Increase with **guess_max**.
read_sheet(path, **guess_max** = Inf)

Set all columns to same type, e.g. character
read_sheet(path, col_types = "c")

Set each column individually
# col types: skip, guess, integer, logical, character
read_sheets(ss, col_types = "_?ilc")

COLUMN TYPES

l	n	c	D	L
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

- skip - "_" or "-"
- guess - "?"
- logical - "l"
- integer - "i"
- double - "d"
- numeric - "n"
- date - "D"
- datetime - "T"
- character - "c"
- list-column - "L"
- cell - "C" Returns list of raw cell data.

Use list for columns that include multiple data types. See **tidyr** and **purrr** for list-column data.

OTHER USEFUL EXCEL PACKAGES

For functions to write data to Excel files, see:
- **openxlsx**
- **writexl**

For working with non-tabular Excel data, see:
- **tidyxl**

CELL SPECIFICATION FOR READXL AND GOOGLESHEETS4

Use the **range** argument of **readxl::read_excel()** or **googlesheets4::read_sheet()** to read a subset of cells from a sheet.
read_excel(path, range = "Sheet1!B1:D2")
read_sheet(ss, range = "B1:D2")

Also use the range argument with cell specification functions **cell_limits()**, **cell_rows()**, **cell_cols()**, and **anchored().**

FILE LEVEL OPERATIONS

googlesheets4 also offers ways to modify other aspects of Sheets (e.g. freeze rows, set column width, manage (work)sheets). Go to **googlesheets4.tidyverse.org** to read more.

For whole-file operations (e.g. renaming, sharing, placing within a folder), see the tidyverse package **googledrive** at **googledrive.tidyverse.org**.

Package `dplyr`: data manipulation

- **`dplyr`** is very popular package in the R community mostly due to it greatly facilitating the manipulation of data
- Some of its features include:
  - the most basic data manipulation operations are implemented;
  - handles multiple types of data structures (e.g. data frames, databases, ...);
  - but mostly, it's fast!

Package `dplyr`: data manipulation (cont.)

- **`tibble`**: a data frame table specifically tailored for the operations of `dplyr`;

```
library(dplyr)
data(iris)
ir <- as_tibble(iris)
glimpse(ir)
```

```
## Rows: 150
## Columns: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4
## $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1
## $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa,
```

Package `dplyr`: basic operations

```
ds2 <- <operation>(ds1,...)
```

- **filter** : select a subset of rows
- **select** : select a subset of columns
- **arrange** : reorder the rows
- **mutate** : generate new columns
- **summarize** : summarize column values

These basic operations return a new object following the intended operation.
They do not change the object in the first parameter (the tibble)

Package `dplyr`: filter

`filter(ds1, cond1, cond2, ...)` returns the rows of the tibble `ds1` satisfying all the conditions `cond1`, `cond2`, ...

```
filter(ir, Sepal.Length > 6, Sepal.Width > 3.5)
```

```
## # A tibble: 3 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          7.2         3.6          6.1         2.5 virginica
## 2          7.7         3.8          6.7         2.2 virginica
## 3          7.9         3.8          6.4         2   virginica
```

```
filter(ir, Sepal.Length > 7.7 | Sepal.Length < 4.4)
```

```
## # A tibble: 2 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          4.3         3            1.1         0.1 setosa
## 2          7.9         3.8          6.4         2   virginica
```

`select(ds1, col1, col2, ...)` returns the columns `col1`, `col2`, ... of the tibble `ds1`

```r
select(ir, Sepal.Length, Species)
```

```
## # A tibble: 150 x 2
##    Sepal.Length Species
##           <dbl> <fct>
##  1          5.1 setosa
##  2          4.9 setosa
##  3          4.7 setosa
##  4          4.6 setosa
##  5          5   setosa
##  6          5.4 setosa
##  7          4.6 setosa
....
```

Package dplyr: select (cont.)

You can use select in a *negative* way, passing information concerning the columns that the user does not want to select.

```
select(ir, -(Sepal.Length:Petal.Length))
```

```
## # A tibble: 150 x 2
##    Petal.Width Species
##          <dbl> <fct>
##  1         0.2 setosa
##  2         0.2 setosa
##  3         0.2 setosa
##  4         0.2 setosa
##  5         0.2 setosa
##  6         0.4 setosa
##  7         0.3 setosa
....
```

Package dplyr: select (cont.)

If you have a certain number of variables that begin with the same name, you can select them all easily

```
select(ir, starts_with("Sepal"))
```

```
## # A tibble: 150 x 2
##    Sepal.Length Sepal.Width
##           <dbl>       <dbl>
##  1          5.1         3.5
##  2          4.9         3  
##  3          4.7         3.2
##  4          4.6         3.1
##  5          5           3.6
##  6          5.4         3.9
##  7          4.6         3.4
....
```

Package dplyr: arrange

`arrange(ds1, col1, col2, ...)` re-arranges the rows of the tibble `ds1` by the user input order (`col1`, `col2`, ...)

```
arrange(ir, desc(Sepal.Length), Sepal.Width)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          7.9         3.8          6.4         2   virginica
##  2          7.7         2.6          6.9         2.3 virginica
##  3          7.7         2.8          6.7         2   virginica
##  4          7.7         3            6.1         2.3 virginica
##  5          7.7         3.8          6.7         2.2 virginica
##  6          7.6         3            6.6         2.1 virginica
##  7          7.4         2.8          6.1         1.9 virginica
....
```

`mutate(ds1, newcol1, newcol2, ...)` adds new columns (`newcol1`, `newcol2`, `...`) to the tibble `ds1`. It does not change the original data.

```
mutate(ir, sr = Sepal.Length/Sepal.Width, pr = Petal.Length/Petal.Width)
```

```
## # A tibble: 150 x 7
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species    sr    pr
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>   <dbl> <dbl>
##  1          5.1         3.5          1.4         0.2 setosa   1.46  7
##  2          4.9         3            1.4         0.2 setosa   1.63  7
##  3          4.7         3.2          1.3         0.2 setosa   1.47  6.5
##  4          4.6         3.1          1.5         0.2 setosa   1.48  7.5
##  5          5           3.6          1.4         0.2 setosa   1.39  7
##  6          5.4         3.9          1.7         0.4 setosa   1.38  4.25
##  7          4.6         3.4          1.4         0.3 setosa   1.35  4.67
....
```

Package `dplyr`: summarize

`summarize(ds1, sumF1, sumF2, ...)` summarizes the rows in the tibble `data` using the user-provided functions `sumF1`, `sumF2`, ...

```
summarise(ir, avgPL = mean(Petal.Length), varSW = var(Sepal.Width))
```

```
## # A tibble: 1 x 2
##   avgPL varSW
##   <dbl> <dbl>
## 1  3.76 0.190
```

Package `dplyr`: combining operations

`dplyr` allows for the combination of basic operations in the same call

```
select(filter(ir, Petal.Width > 2.3), Sepal.Length, Species)
```

```
## # A tibble: 6 x 2
##   Sepal.Length Species
##          <dbl> <fct>
## 1          6.3 virginica
## 2          7.2 virginica
## 3          5.8 virginica
## 4          6.3 virginica
## 5          6.7 virginica
## 6          6.7 virginica
```

Package dplyr: combining operations (cont.)

However, composing such functions can become very hard to understand (and code...)

```
arrange(select(filter(mutate(ir, sr = Sepal.Length/Sepal.Width),
    sr > 1.6), Sepal.Length, Species), Species, desc(Sepal.Length))
```

```
## # A tibble: 103 x 2
##    Sepal.Length Species
##           <dbl> <fct>
##   1          5   setosa
##   2          4.9 setosa
##   3          4.5 setosa
##   4          7   versicolor
##   5          6.9 versicolor
##   6          6.8 versicolor
##   7          6.7 versicolor
....
```

Package `dplyr`: chaining operator

- To provide an easy solution for the combination of `dplyr` operations, one can use the chaining operator (or pipe) **`%>%`**
- If using this operator, you only need to declare the `tibble` in the first function call
- The chaining operator tells the following operation that the result of the former operation will be the `tibble` to use
- `x %>% f(y)` becomes `f(x,y)`

Package dplyr: chaining operator (cont.)

```r
mutate(ir, sr = Sepal.Length/Sepal.Width) %>%
    filter(sr > 1.6) %>%
    select(Sepal.Length, Species) %>%
    arrange(Species, desc(Sepal.Length))
```

```
## # A tibble: 103 x 2
##    Sepal.Length Species
##           <dbl> <fct>
##  1          5   setosa
##  2          4.9 setosa
##  3          4.5 setosa
##  4          7   versicolor
##  5          6.9 versicolor
##  6          6.8 versicolor
##  7          6.7 versicolor
##  8          6.7 versicolor
##  9          6.7 versicolor
## 10          6.6 versicolor
## # ... with 93 more rows
```

group_by(ds1, crit1, crit2, ...) groups rows of the tibble ds1 according to user-input criteria crit1, crit2, ...

```
sps <- group_by(ir, Species)
sps
```

```
## # A tibble: 150 x 5
## # Groups:   Species [3]
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
....
```

Package dplyr: group_by (cont.)

You can apply summarize to sub-groups.

```
group_by(ir, Species) %>%
    summarise(mPL = mean(Petal.Length))
```

```
## # A tibble: 3 x 2
##   Species      mPL
##   <fct>      <dbl>
## 1 setosa      1.46
## 2 versicolor  4.26
## 3 virginica   5.55
```

Package `tidyr`: basic operations

- **`complete`**: make implicit missing values explicit
- **`drop_na`**: make explicit missing values implicit
- **`fill`**: replace missing values with next/previous value
- **`replace_na`**: replace missing values with a known value
- **`pivot_longer`**: “lengthens” data, increasing the number of rows and decreasing the number of columns.
- **`pivot_wider`**: “widens” data, increasing the number of columns and decreasing the number of rows.

Package tidyr: basic operations (cont.)

```r
df <- tibble(x = c(1, 2, NA), y = c("a", NA, "b"))
df
```

```
## # A tibble: 3 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     2 <NA>
## 3    NA b
```

```r
df %>%
    drop_na()
```

```
## # A tibble: 1 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
```

```r
df %>%
    drop_na(x)
```

```
## # A tibble: 2 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     2 <NA>
```

```r
df %>%
    replace_na(list(x = 0,
        y = "?"))
```

```
## # A tibble: 3 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     2 ?
## 3     0 b
```

Package `tidyr`: basic operations (cont.)

Dataset: Pew religion and income survey with nr of respondees with an income range

```
relig_income
```

```
## # A tibble: 18 x 5
##    religion                `<$10k` `$10-20k` `$20-30k` `$30-40k`
##    <chr>                     <dbl>     <dbl>     <dbl>     <dbl>
##  1 Agnostic                     27        34        60        81
##  2 Atheist                      12        27        37        52
##  3 Buddhist                     27        21        30        34
##  4 Catholic                    418       617       732       670
##  5 Don’t know/refused           15        14        15        11
##  6 Evangelical Prot            575       869      1064       982
##  7 Hindu                         1         9         7         9
##  8 Historically Black Prot     228       244       236       238
##  9 Jehovah's Witness            20        27        24        24
## 10 Jewish                       19        19        25        25
## 11 Mainline Prot               289       495       619       655
## 12 Mormon                       29        40        48        51
....
```

Package tidyr: basic operations (cont.)

```
relig_income %>%
    pivot_longer(-religion, names_to = "income", values_to = "count")
```

```
## # A tibble: 72 x 3
##    religion income  count
##    <chr>    <chr>   <dbl>
##  1 Agnostic <$10k      27
##  2 Agnostic $10-20k    34
##  3 Agnostic $20-30k    60
##  4 Agnostic $30-40k    81
##  5 Atheist  <$10k      12
##  6 Atheist  $10-20k    27
##  7 Atheist  $20-30k    37
....
```

Package `tidyr`: basic operations (cont.)

Dataset: 2017 American Community Survey with median yearly income and median monthly rent estimates and margin of error

```
us_rent_income
```

```
## # A tibble: 104 x 5
##    GEOID NAME       variable estimate   moe
##    <chr> <chr>      <chr>       <dbl> <dbl>
##  1 01    Alabama    income      24476   136
##  2 01    Alabama    rent          747     3
##  3 02    Alaska     income      32940   508
##  4 02    Alaska     rent         1200    13
##  5 04    Arizona    income      27517   148
##  6 04    Arizona    rent          972     4
##  7 05    Arkansas   income      23789   165
....
```

```
us_rent_income %>%
    pivot_wider(names_from = variable, values_from = c(estimate, moe))
```

```
## # A tibble: 52 x 6
##    GEOID NAME                 estimate_income estimate_rent moe_income moe_re
##    <chr> <chr>                          <dbl>         <dbl>      <dbl>    <db
##  1 01    Alabama                        24476           747        136
##  2 02    Alaska                         32940          1200        508
##  3 04    Arizona                        27517           972        148
##  4 05    Arkansas                       23789           709        165
##  5 06    California                     29454          1358        109
##  6 08    Colorado                       32401          1125        109
##  7 09    Connecticut                    35326          1123        195
##  8 10    Delaware                       31560          1076        247
##  9 11    District of Columbia           43198          1424        681
## 10 12    Florida                        25952          1077         70
## # ... with 42 more rows
```

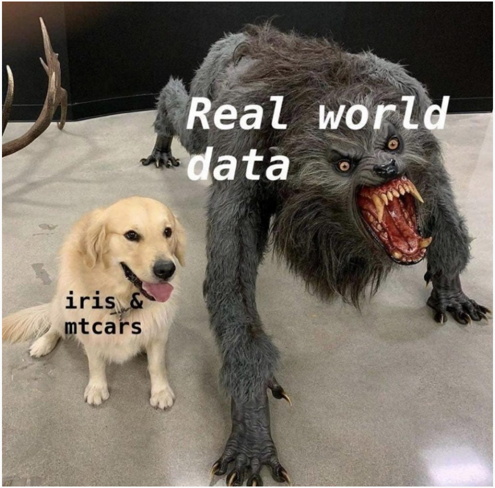
- `%>%` is the chaining operator or pipe: `x %>% f(y)` becomes `f(x,y)`
- `.` represents the previous value in the chain, i.e. `x %>% f(.)` becomes `f(x)`
- `~` is used for anonymous functions. i.e. `function(x) x + 2` can be written as
  - `~ .x + 2`, where `.x` represents the first argument of the function
  - `~ . + 2`, in case the first argument is the previous value in the chain

R references for Data Wrangling

- R for Data Science, Hadley Wickham and Garrett Grolemund (2017)
- More details on these packages from `tidyverse` and other packages: RStudio Cheatsheets

Data Quality

Why?



Data Quality

- The raw format of real data is usually widely variable as values may be missing, insconsistent across different data sources, erroneous.
- Poor data quality poses several challenges to the effective data analysis

Example:

- A classification model for predicting a client's loan risks is built using poor data
  - credit-worthy candidates are denied loans
  - loans are given to individuals that default

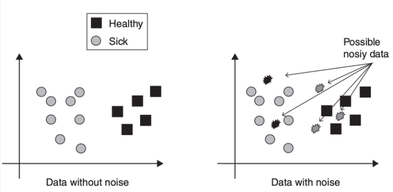
Data Quality (cont.)

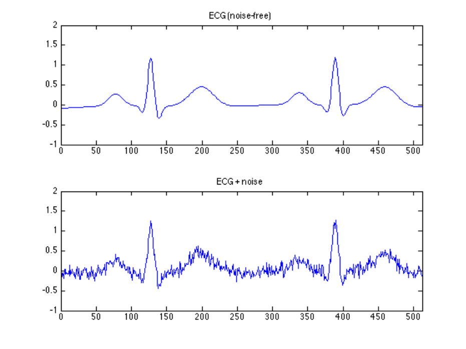
- What are the kinds of data quality problems?
- How can we detect problems with the data?
- What can we do about these problems?

- Examples of data quality problems:
  - Noise and outliers
  - Missing values
  - Duplicate data
  - Incosistent or incorrect data

Noise

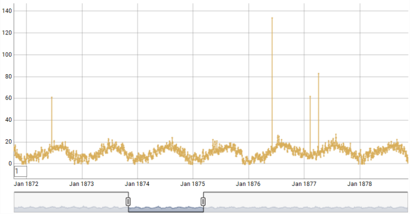
- Noise may refer to irrelevant or useless information
- It can be caused by incorrect or distorted measurements
- It can also be caused by the proper variability of the domain





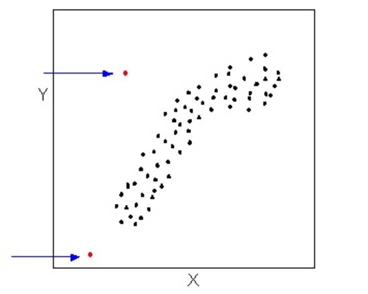
Outliers

- Outliers are data objects with characteristics that are considerably different than most of the other data objects in the data set
- Case 1: outliers are noise that interferes with data analysis
  - $130^\circ C$ value for air temperature



Outliers (cont.)

- Case 2: outliers are the goal of our analysis
  - credit card fraud, intrusion detection



- What are the causes?

Missing Values

- Missing Completely at Random (MCAR)
  - missing value is independent of observed and unobserved data
  - there is nothing systematic about it
  - e.g. a lab value because a lab sample was processed improperly
- Missing at Random (MAR)
  - missing value is related to observed data, not to unobserved data.
  - there may be something systematic about it
  - e.g. missing income value may depend on the age
- Missing Not at Random (MNAR)
  - missing value is related to unobserved data of the variable itself
  - informative / non-ignorable missingness
  - e.g. a person did not entered his/her weight in a survey

Missing Values (cont.)

Solutions:

- remove observations with missing values, i.e. consider only complete cases
  - critical if there are meany observations with missing values
- ignore missining values in the analytical phase
  - use methods that are inherently designed to work robustly with missing values
- make estimates to fill the missing values - imputation
  - the most common value of the attribute (e.g. mean, mode); based on other(s) attribute(s); more sophisticated methods
  - it might introduce bias in data and affect the results

Duplicates

- Data set may include data objects that are duplicates, or almost duplicates of one another
  - Major issue when merging data from heterogeneous sources
- Examples
  - Same person with multiple email addresses
- It is necessary a process of dealing with duplicate data issues
  - When should duplicate data not be removed?

Inconsistent or Incorrect Data

- This the hardest type of data quality issues to detect
- It may depend on expert domain knowlege
- Examples:
  - 4/11/2000: Nov. 4th or April, 11th?
  - author name in a publication (e.g. John Smith, J. Smith, Smith J.)
  - a city called Shanghai in the United States

References

References

Aggarwal, Charu C. 2015. *Data Mining, the Texbook*. Springer.

Gama, João, André Carlos Ponce de Leon Ferreira de Carvalho, Katti Faceli, Ana Carolina Lorena, and Márcia Oliveira. 2015. *Extração de Conhecimento de Dados: Data Mining -3rd Edition*. Edições Sílabo.

Gandomi, Amir, and Murtaza Haider. 2015. "Beyond the Hype: Big Data Concepts, Methods, and Analytics." *International Journal of Information Management* 35 (2): 137–44. doi:https://doi.org/10.1016/j.ijinfomgt.2014.10.007.

Han, Jiawei, Micheline Kamber, and Jian Pei. 2011. *Data Mining: Concepts and Techniques*. 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Moreira, João, Andre Carvalho, and Tomás Horvath. 2018. *Data Analytics: A General Introduction*. Wiley.

"R Project." 2021. https://www.r-project.org/.

Tan, Pang-Ning, Michael Steinbach, Anuj Karpatne, and Vipin Kumar. 2018. *Introduction to Data Mining*. 2nd ed. Pearson.